Tools

These are our tools: - ltrace - strings - file - objdump - gdb

This is a great little trick. If you are working a lot with hexadecimal and you want to easily convert it to ascii you can write this in bash

```
echo 6a6548 | xxd -r -p
```

will print out: **jeH** xxd is a program that makes ascii into hexdumps. with the -r we can reverse it.

Nasm to opcode

If we want to convert an assembly instruction to opcode we can use this tool

```
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb

nasm > JMP ESP
00000000  FFE4              jmp esp
```

Objdump

Objdump is a program that outputs the assembly code of a compiled program. It ca be executed like this. example:

```
objdump -D myProgram
objdump -M intel -d program_name ;This is to read the assembly in intel-syntax
```

GDB - GNU Debugger

Setting breakpoints

Sometimes you want the debugger to stop at a certain point in the program, so that you can investigate memory and stuff. We can set these breakpoints with the following command:

Set a break at the main-function

```
break main
```

Set a break at that line. I think it is set before the line is executed.

```
break 10
```

Show breakpoints If you want to know which breakpoints you have set you can run:

```
info breakpoint
info break
info b
```

Remove breakpoints Will delete all breakpoints on line 9

```
clear 9
```

Run the program

```
run
```

Show code

```
list ; show code if you have compile it with the -g flag
list 10` ; will show the code around line 10. five lines before, and five lines after.
list 1,20 ; will list all lines between the numbers.
```

This shows the code in assembly. It is pretty much the same as running objdump.

```
disassemble main
```

Show info about instruct pointer

```
info register eip
i r eip
```

On 64bit machines it is called `rip` instead of `eip`. It basically shows to what address eip is pointing at. So the output might be something like this:

```
(gdb) i r rip
rip            0x4004aa 0x4004aa <main+4>
```

Which means that rip at this moment is pointing at 0x4004aa. Which means that this is going to be the next instruct that gets executed.

The structure is like this:

```
examine/[format] address
x/
```

Format is how you want to display the memory. Here are the following formats:

```
    o - octal
    x - hexadecimal
    d - decimal
    u - unsigned decimal
    t - binary
    f - floating point
    a - address
    c - char
    s - string
    i - instruction
```

Example:

```
x/s myVariable
```

This means: examine myVariable, and output the content in that memory in the form of a string. Now this does not work for values that does not have a memory address. It will just give you `

```
0x16:   Cannot access memory at address 0x16
```

That is because the variable is not a pointer (it does not point to an memory-address), but instead it is a hardcoded value.

```
x/i $rip
```

Examine/info instruction pointer register. This command can be used to examine a specific part of memory. In this example it was the instruct pointer, but it can also be a specific address in memory.

Show all functions

```
info funcions
```

Python Python can be quite useful go generate strings as input. Of course this can be done with a lot of other languages. so it would work like this.

```
./myProgram $(python -c 'print "\x41" * 30')
```

Basically, the `$(python)` creates a shell within our command. And in that shell we run the normal python command. The `-c` flag tells python that we are going to run a command instead of opening up the interactive shell. You can test this in the terminal like this:

```
$ python -c 'print "hello" * 100'
hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohe
```

GCC

Compile the program in debugger mode, so that the debugger has access to the code.

```
gcc -g program.c
```